

PROGRAMMING Expert

Having trouble keeping track of your usernames and passwords? Mike James can help you crack the code. Here he explains how to avoid being locked out of your favourite sites with the help of PassKey



C# AND VISUAL BASIC FOR FREE

Microsoft has promised to make all the Visual Studio Express products available for free download for a year from the launch of the full Visual Studio 2005 development system. This means you can download free versions of C#, VB, Web Developer, C++, J# and SQL Server until 6th November 2006. Before the launch, Microsoft said it intended to charge \$49 (around £28) for the Express product line, except SQL Server 2005 Express Edition, which was always intended as a free version of SQL Server 2005. The idea is to boost the use of the .NET languages by enthusiasts and encourage upgrades in the workplace. For details, visit <http://msdn.microsoft.com/vstudio/express>.

EBAY API BARGAIN

Perhaps the best eBay bargain of all has been announced: the eBay API is free to use. Before you had to pay from \$1.25 to \$2.90 (71p to £1.65) per 1,000 items listed and an annual fee of \$500 (around £285). You can use the API to write applications that help people buy or sell items on eBay. As further encouragement, eBay has also set up a developer contest with a top prize of \$5,000 (around £2,849) for the most innovative eBay application. See <http://developer.ebay.com> for more details.

STUDIO 11 FREE DOWNLOAD

Sun has made Studio 11 free to download. It provides development facilities for mixed-language programming in Java, C, C++ and Fortran. However, its system requirements are Solaris or Linux running on Sparc or Intel hardware. It used to sell for \$3,000 (around £1,709) per copy. For more details, go to www.sun.com/software/products/studio.



A constant headache of the internet age is remembering usernames and passwords. It's not safe to use the same username everywhere, and many websites have restrictions on valid usernames, which forces you to use several different variants. As for passwords, we all know the strongest passwords are meaningless combinations of letters and numbers, but who can remember them? And even if you opt for memorable but weak passwords, you're forever being locked out because you've forgotten which variation on your preferred username the site graciously allowed you to use.

Though 'single sign-on' services such as Microsoft's Passport address this problem, take-up by websites has so far been limited. It looks as if we're stuck with the username and password mess and we need to make the best of it. That's the theme of this month's PassKey project. We decided it was time to build a single password-protected utility that can automate logons to all the websites we use. Here we have implemented PassKey as a Windows application rather than a website, simply to make sure the password data is secure.

Programming topics encountered in this month's project include creating custom dialog boxes, verifying passwords without storing them, working with isolated storage, reading and writing encrypted files, connecting streams, using the DataGridView control unbound, using VB facilities in C#, creating a controlled delay, and running and communicating with other programs.

GETTING THE PASSWORD

PassKey needs to be password-protected, otherwise anyone with access to your PC will be able to use all your passwords. Rather than require the user to provide a username and password, PassKey assumes

that the user is the one currently logged into Windows. To access another user's passwords, they must log in to Windows first. This provides a level of security. To make it more secure, though, we will demand a password immediately when PassKey is run, using a custom dialog box.

Start a new Windows Form project called PassKeys. Add to it a new form called PassWordDlg. Set FormBorderStyle to FixedDialog, and set ControlBox, MinimizeBox and MaximizeBox to false. Add two buttons and a text box. Call one button OK and the other Cancel, and call the text box PassWordBox. If you want to hide the password as it is typed in, you can set the text box's PasswordChar property to an asterisk or any other character to obscure the entry. Your dialog should look something like the image on page 263.

The .NET system makes implementing dialog boxes very easy. Each button has an associated DialogResult property that can be used to pass an OK or Cancel result back to the application that spawns the dialog box. Set the OK button's DialogResult property to OK and the Cancel button's to Cancel.

To allow our application to access the password entered in the dialog box, we must create a property that the application can check. A simple string property will do the job. Within the class definition of PassWordDlg, add:

```
public string Password
{
    get
    {
        return PassWordBox.Text;
    }
}
```



We don't need a set method for the property. The main form can use the password dialog box by creating an instance and then using the DialogResult and Password properties:

```
private void Form1_Load(
    object sender,
    EventArgs e)
{
    PasswordDlg PW=new PasswordDlg();
    PW.ShowDialog();
    if (PW.DialogResult ==
        DialogResult.Cancel)
        Application.Exit();
    password = PW.Password;
    PW.Dispose();
}
```

We also need a global variable to store the password:

```
public partial class Form1 : Form
{
    private string password;
```

If the user clicks Cancel, the application comes to an end. You must have the password to proceed.

VALIDATING PASSWORDS

So how do we validate the password entered by the user and give them access to the main application? The obvious answer is check it against a stored file of valid passwords, but this isn't a sensible idea. You should avoid storing passwords on disk unless they are encrypted. A much better solution is to use the supplied password to generate an encryption key and see if it is possible to read an encrypted file using this key. If it is, the user has supplied the correct password. This kills two birds with one stone as we would need to encrypt all the passwords stored by PassKey anyway. By generating a key from the password, we can decrypt the data and validate the password at the same time.

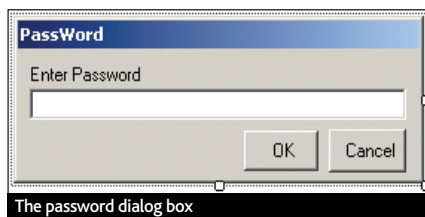
The best way to do this is to create a new class. Add a class called CProfile to the project. This will accept the password and turn it into a key that is suitable for encryption use:

```
public CProfile(string password)
{
    m_password = password;
    PasswordDeriveBytes PassBytes =
        new PasswordDeriveBytes(
            password, null);
    key = PassBytes.CryptDeriveKey(
        "RC2", "SHA1", 128,
        new byte[] { 0, 0, 0, 0, 0, 0,
            0, 0 });
```

The 128-bit key is created from the password using the SHA1 method so as to be suitable for the RC2 encryption method. We haven't got room to discuss the workings of these encryption methods, but both produce reasonably strong encryption.

We also need to define the key as a private global variable and add a using statement to the start of the class. To make things simpler, we might as well add all the necessary using and global variables at this point:

```
using System.IO.IsolatedStorage;
using System.IO;
```



```
using System.Security.Cryptography;
using System.Windows.Forms;
namespace PassKey
{
    class CProfile
    {
        private byte[] key;
        private RC2CryptoServiceProvider
            rc2;
        private IsolatedStorageFile
            IsoStore;
        private IsolatedStorageFileStream
            IsoStream;
        private CryptoStream cs;
        private StreamReader sr;
        private StreamWriter sw;
        private string recPass;
        private string m_password;
```

Now we have a key, we can create the crypto object that implements the RC2 algorithm:

```
rc2 = new RC2CryptoServiceProvider();
rc2.Key = key;
rc2.IV = new byte[] { 21, 22,
    23, 24, 25, 26, 27, 28 };
```

The IV or initial value property should be set to a random value, but the same random value each time. In this case, setting it to a group of fixed values is just as secure.

If the encrypted file already exists, we can attempt to read it. We need something at the start that can be used to verify successful decryption. We could encrypt a word such as 'success' and look for this in the decrypted file, but using any real word would make the hacker's job easier. A better way is to encrypt the password itself and then try to recover it using the password supplied by the user.

The next issue is where to store the file. There is always a problem in setting up files to store custom application data without running into files or directories of the same name that exist already. The .NET system provides a solution to this and wider problems with its isolated storage facility. This provides an area of storage that is unique to the current user and the application:

```
IsoStore =
    IsolatedStorageFile.GetStore(
        IsolatedStorageScope.User |
        IsolatedStorageScope.Domain |
        IsolatedStorageScope.Assembly,
        null, null);
```

Now we have an IsoStore object that is unique to the user, domain and assembly – that is, the user and the program. This will be a folder created in Documents and Settings under the user's name. You can discover exactly where the file is by examining the IsoStore object in the debug window. If the file doesn't exist, we have to create it using the new password that the user has just supplied. You could

give the file a descriptive name such as 'secret data.dat', but why make things easy for hackers? Defining the filename as a constant makes it possible to change it:

```
private const string
    profile = @"\IDQ756.ATX";
```

This name doesn't give much away about what it's for. Next we check to see if it exists, and create it if it doesn't:

```
if (IsoStore.GetFileNames(profile)
    .Length == 0)
{
    sw = OpenWrite();
    Close();
}
```

The OpenWrite function (which we have yet to write) returns a StreamWriter object that can be used to write text to the encrypted file. The function also writes the encrypted password to the start of the file.

Once the file exists, because we just created it using the new password or created it previously, we can open it, read the stored password into recPass and compare it to the supplied password:

```
sr=OpenRead();
Close();
if (recPass != m_password)
    Application.Exit();
}
```

The OpenRead function reads the password automatically as part of opening the file. If the passwords don't match, we just stop the application. You could add something that tells the user they've provided the wrong password if you want. The OpenRead function also returns a StreamReader object that allows the encrypted file to be read as text.

OPENING STREAMS

We haven't yet tackled the problem of opening the encrypted streams. Many of the cryptographic examples that you find don't make good use of streams. In .NET, you can think of a stream as a sequence of data items. Some streams are simple byte streams and others are text or character streams. As long as it makes sense, you should be able to connect streams together.

In the OpenWrite function, we create an IsolatedStorageFileStream:

```
public StreamWriter OpenWrite()
{
    IsoStream =
        new IsolatedStorageFileStream(
            profile,
            FileMode.Create,
            FileAccess.Write,
            IsoStore);
```

This is a byte stream that can be used to write raw bytes to a file in the isolated storage just created. We can then connect a CryptoStream to the file stream to feed it with encrypted bytes:

```
cs = new CryptoStream(
    IsoStream,
```



REVIEW BOOK

Sockets, Shellcode, Porting and Coding

RATING ★★

PRICE £21

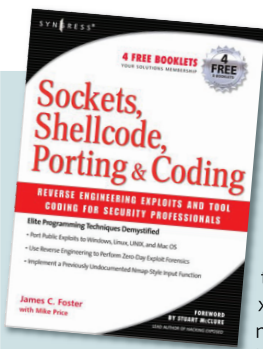
SUPPLIER www.amazon.co.uk

ISBN 1597490059

PAGES 700

This book is about hacking, but you may not guess it from the title or subtitle, *Reverse Engineering Exploits and Tool Coding for Security Professionals*. You might not even guess if you look inside.

Author James Foster starts with a slight and strange discussion of C, C++, C#, Java, Perl and Python. You'll need to know a lot more about these languages than this book tells you. The same is true of most of the book, containing as it does long, rambling explanations of what should be obvious to any trainee hacker.



A lot of the content seems to be presented to pad the book out to a reasonable number of pages. Occasionally it reads as if the author has lived on Mars for a few years and has just encountered some of the technologies. But then we dive into x86 assembler as if it were second nature. The author seems to be much happier with low-level techniques than new high-level technologies.

The book aims to make people aware of the threats hackers pose. However, those interested in Windows should know much of the discussion is about UNIX or Linux. And while the book provides the occasional insight into how hacking works, you can easily disregard 90 per cent of it.



One or two really worthwhile insights into hacking



Lots of irrelevant padding

```
public void Close()
{
    if (sw != null)
    {
        sw.Close();
        sw=null;
    }
    if (sr != null)
    {
        sr.Close();
        sr=null;
    }
    if (cs != null)
    {
        cs.Close();
        cs=null;
    };
    if (IsoStream != null)
    {
        IsoStream.Close();
        IsoStream = null;
    }
}
```

CHECKING THE PASSWORD

Now that we have the password and encryption implemented, it is time to return to the main part of the program. When the main form loads, it currently asks for the password, and now we have the means to check it. Before doing this, we need to add a control that can hold the details that are stored in the file. Add a dataGridView control and add columns for Site, URL, username and password. You can add additional columns and generally customise and improve the user interface later. Also, add a button with the title Add/Edit.

Now we can add to the end of the FormLoad event handler:

```
profile = new CProfile(password);
ReadData();
```

We also need a global variable to keep track of the profile object:

```
private CProfile profile;
```

Creating the profile object does more than it appears because it also checks the password against the one in the file. If they match, the program moves on and reads the data stored in the file into the grid. It is more logical to move on and consider the WriteData function first because you can't read the data until you know the format used to write it.

To use the program the user clicks the Add/Edit button, changes or adds data to the grid and when they click the button again it is written to the encrypted file:

```
private void button1_Click(
    object sender,
    EventArgs e)
{
    if (editing)
    {
        editing = false;
        button1.Text = "Add/Edit";
        WriteData();
    }
    else
    {

```

REVIEW BOOK

The Robosapien Companion

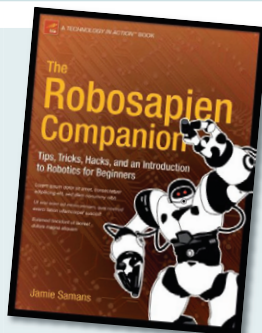
RATING ★★★★★

PRICE £13

SUPPLIER www.amazon.co.uk

ISBN 1590595262

PAGES 328



DIY robots are a great way to learn about programming and electronics, but starting from scratch can be tough. Robosapien is a toy many gadget-oriented readers may have heard about. Jamie Samans' book goes a long way towards turning a cool toy into an fun learning experience.

The book explains the history of the Robosapien, which is an example of a Biology, Electronics, Aesthetics and Mechanics (BEAM) design robot, as well as its mechanics and software. It isn't a book for the expert, but if you

are an expert you'll still want a copy – because it will save you time if you intend to use the device to do something different. This is an enthusiast's book written by an enthusiast. But if you are unhappy about taking a drill or saw to your toy, be warned: this book takes the word 'hacking' to a whole new level.

The final chapter deals briefly with newer models of BEAM robots on the market. The appendix includes an interview with Mark Tilden, the designer of Robosapien and the inventor of the BEAM approach to robotics.

If you want more fun with your toy or you want to know if it's worth buying a Robosapien, this book is ideal. Highly recommended.



An exciting introduction to the hardware and software of robotics



You have to take a drill or saw to your Robosapien

```
rc2.CreateEncryptor(),
CryptoStreamMode.Write);
```

This is also a byte stream. It accepts bytes, encrypts them using the RC2 method created earlier and then passes them along to the IsoStream to be written to the file. We could stop at this point, but we need to write strings to the crypto stream so let's create one more stream, a StreamWriter connected to the CryptoStream:

```
sw = new StreamWriter(cs);
```

All that remains is to write the password to the file and return the StreamWriter:

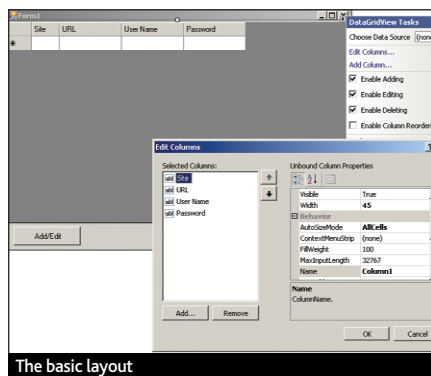
```
sw.WriteLine(m_password);
return sw;
```

When we send strings to the StreamWriter, it converts them to bytes, which the CryptoStream encodes, and finally the IsoStream writes to the file.

As you can guess, the OpenRead function works in the same way and uses the same three types of stream:

```
public StreamReader OpenRead()
{
    IsoStream =
        new IsolatedStorageFileStream(
            profile,
            FileMode.Open,
            FileAccess.Read,
            IsoStore);
    cs = new CryptoStream(
        IsoStream,
        rc2.CreateDecryptor(),
        CryptoStreamMode.Read);
    sr=new StreamReader(cs);
    recPass = sr.ReadLine();
    return sr;
}
```

The Close function is simply:



The basic layout

```
editing = true;
button1.Text = "Finish";
}
}
```

The program changes the button's text according to whether or not it is in editing mode. We also need a global variable to keep track of the mode:

```
private bool editing = false;
```

The WriteData function is easy. It simply opens the file and writes the current contents of the grid in row order:

```
private void WriteData()
{
    StreamWriter sw= profile.OpenWrite();
    int rows = dataGridView1.Rows.Count - 2;
    sw.WriteLine(rows.ToString());
    int cols = dataGridView1.Columns.Count - 1;
    sw.WriteLine(cols.ToString());
    for(int i=0;i<= rows;i++)
    {
        for(int j=0;j<=cols;j++)
        {
            sw.WriteLine(dataGridView1.Rows[i].Cells[j].Value);
        }
    }
    profile.Close();
}
```

The first two values stored in the file are the number of rows and columns to be stored. This is used by the ReadData function to discover how much data to read:

```
private void ReadData()
{
    StreamReader sr = profile.OpenRead();
    if (sr.EndOfStream)
    {
        profile.Close();
        return;
    }
    int rows = Convert.ToInt32(
        sr.ReadLine());
    int cols = Convert.ToInt32(
        sr.ReadLine());
    for (int i = 0; i <= rows; i++)
    {
        dataGridView1.Rows.Add();
        for (int j = 0; j <= cols; j++)
        {
```

```
            dataGridView1.Rows[i].Cells[j].
            Value = sr.ReadLine();
        }
    }
    profile.Close();
}
```

To use streams, we need to add to the start of file:

```
using System.IO;
```

BORROWING FROM VISUAL BASIC

With these two functions complete, we have a program that allows the user to enter data, and save it to or load it from an encrypted file. We need the part of the program that enters this data into websites. The first problem is to start Internet Explorer and navigate to the URL in the table. We want this to happen when the user clicks on the site name in the Site column. Visual Basic provides the Shell method and the SendKeys command to open applications and interact with them. But if you look at the documentation or search online for help, you'll see statements to the effect that this cannot be done in C#, because it has no managed equivalent of the VB commands. However, there is nothing to say that C# cannot use a class intended for VB. To do this, we need to add:

```
using Microsoft.VisualBasic;
```

Add a reference to Microsoft.VisualBasic using the menu command Project, Add Reference. After this, we can use the Interaction class (which has a Shell method) and the SendKeys class in C#. We can write the dataGridView1's CellClick event handler.

We want to ensure that nothing happens if the grid is in edit mode:

```
private void dataGridView1_CellClick(
    object sender,
    DataGridViewCellEventArgs e)
{
    if (editing) return;
```

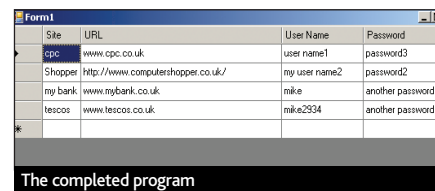
If it isn't in edit mode, we start Internet Explorer only if the click is in the first column. If it is, we retrieve the URL from the same row and start IE with this as a command parameter:

```
if (e.ColumnIndex == 0)
{
    String IE = @"C:\Program Files\
    Internet Explorer\IEXPLORE.EXE";
    String URL =(String)dataGridView1.
    Rows[e.RowIndex].Cells[e.
    ColumnIndex + 1].Value;
    IEinst = Interaction.Shell(
    IE + " " + URL,
    AppWinStyle.NormalFocus,
    false, 0);
}
```

Make sure the string IE gives the location of iexplore.exe for your computer. A global variable is needed to keep track of the instance of Internet Explorer just created:

```
private Int32 IEinst;
```

The IEinst variable stores a 'handle' to the process created and can be used to control it. For example,



The completed program

you can use it to discover if the user has closed the web browser.

The user must wait for the web page to load and click on the username entry box. The user clicks on the appropriate entry in the data grid and the username is transferred to the box. The procedure is repeated for the password and for any information that you care to customise the dataGridView to hold. We need to give the focus back to IE:

```
if (e.ColumnIndex > 1)
{
    Interaction.AppActivate(IEinst);
}
```

In theory, we can just send the data using the SendKeyWait method of the SendKey class. But in practice, it takes a few milliseconds for the instance of IE to become the active window, so we need to pause for a short time. The simplest way to do this is by sending the current thread of execution to sleep using the threading class library. We need:

```
using System.Threading;
```

The command to pause for 100 milliseconds is:

```
Thread.Sleep(100);
```

Finally, the data can be transferred using the SendKeyWait method:

```
SendKeys.SendWait(
    (String)dataGridView1.
    Rows[e.RowIndex].
    Cells[e.ColumnIndex].Value);
}
```

If you try out the program, you'll find you can go automatically to a website and fill in any part of a form simply by clicking on the grid.

IMPROVING PASKEY

The current state of the project is usable in the sense that it all works, but it isn't finished by a long way. Countless improvements can be made. The simplest is to set the form's TopMost property to True to keep the form visible even when the browser has the focus. Work is also needed to keep secret the information in the grid; it should all be blanked out when not in edit mode. Perhaps the user should be asked for the password again before entering edit mode. It should check the instance of Internet Explorer it created is still running before trying to send any data to it. We've put the code of the application as it stands on the cover disc, allowing you to develop it further as you wish. 

NEXT MONTH

GOING FURTHER WITH PASKEY

We show you how to develop PassKey into an even more useful application